Capítulo 5
Modelado de la dinámica

1. Introducción

El objetivo del presente capítulo es explicar de qué manera UML representa las interacciones entre objetos. En el capítulo Conceptos de la orientación a objetos, vimos que los objetos de un sistema poseen su propio comportamiento e interactúan entre sí para dotar al sistema de una dinámica global. En el capítulo Modelado de los requisitos, estudiamos la forma en que los casos de uso representan las acciones y reacciones entre un actor externo y el sistema. Desde el punto de vista del modelado, esos dos tipos de interacciones se distinguen por su diferencia interna/externa, pero no por su naturaleza.

Para responder a la necesidad de representación de las interacciones entre objetos, UML propone dos tipos de diagramas:

– El diagrama de secuencia se centra en aspectos temporales.

– El diagrama de comunicación se centra en la representación espacial.

En el presente capítulo estudiaremos ambos tipos de diagramas. Más tarde examinaremos cómo descubrir progresivamente los objetos que componen un sistema. Dicho descubrimiento se basará en las interacciones entre los objetos que intervienen en los casos de uso del sistema. Para representar las interacciones nos decantaremos por el diagrama de secuencia, ya que suele ser la opción preferida por las personas que se encargan de modelar los proyectos.

2. Diagrama de secuencia

2.1 Introducción

El diagrama de secuencia describe la dinámica del sistema. A menos que se modele un sistema muy pequeño, resulta difícil representar toda la dinámica de un sistema en un único diagrama. Por tanto, la dinámica completa se representará mediante un conjunto de diagramas de secuencia, cada uno de ellos vinculado generalmente a una subfunción del sistema. Estudiaremos los trucos de interacción que facilitan esta posibilidad de representación.

El diagrama de secuencia describe las interacciones entre un grupo de objetos mostrando de forma secuencial los envíos de mensajes entre objetos. El diagrama puede asimismo mostrar las transmisiones de datos intercambiados durante el envío de mensajes.

Observación

Para interactuar entre sí, los objetos se envían mensajes. Durante la recepción de un mensaje, los objetos se vuelven activos y ejecutan el método del mismo nombre. Un envío de mensaje es, por tanto, una llamada a un método.

2.2 Línea de vida de un objeto

Dado que representa la dinámica del sistema, el diagrama de secuencia hace entrar en acción las instancias de clases que intervienen en la realización de la subfunción a la que está vinculado. A cada instancia se asocia una línea de vida que muestra las acciones y reacciones de la misma, así como los periodos durante los cuales ésta está activa, es decir, durante los que ejecuta uno de sus métodos.

La representación gráfica de la línea de vida se ilustra en la figura 5.1.

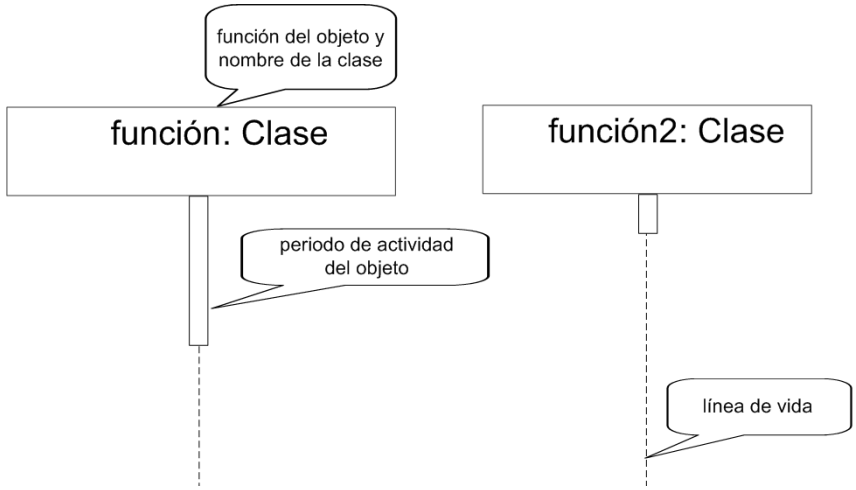


Figura 5.1 - Líneas de vida

Observación

La notación "función: Clase" representa la función de una instancia seguida del nombre de su clase. Para simplificar, en esta obra consideraremos que la función de la instancia corresponde a su nombre. Si sólo una instancia de la clase participa en el diagrama de secuencia, la función de la instancia es opcional. El nombre de la clase puede también omitirse en las etapas preliminares del modelado, pero debe especificarse lo antes posible.

Observación

Los diagramas de secuencia contienen varias líneas de vida, ya que tratan de las interacciones entre varios objetos.

La línea de vida puede empezar con la introducción de un invariante de estado, que es una expresión lógica que debe cumplirse durante todo el desarrollo de la línea de vida. La figura 5.2 ilustra la introducción de dicho invariante.

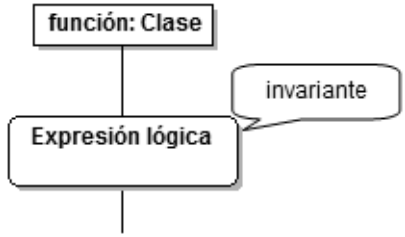


Figura 5.2 - Invariante de estado

2.3 Envío de mensajes

Los envíos de mensajes se representan mediante flechas horizontales que unen la línea de vida del objeto emisor con la línea de vida del objeto destinatario (ver figura 5.3).

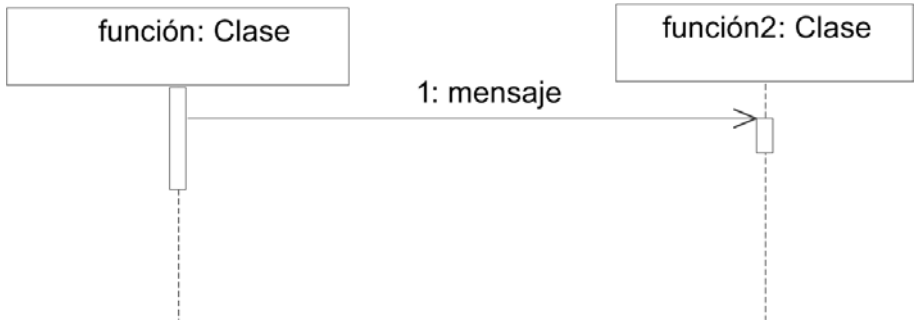


Figura 5.3 - Envío de un mensaje

En la figura 5.3, el objeto de la izquierda envía un mensaje al objeto de la derecha. En programación, este mensaje da lugar a la ejecución del método *mensaje* del objeto de la derecha, lo que provoca su activación. El nombre del mensaje no es obligatorio, es posible omitirlo en la especificación de un envío de mensaje. En este caso, el nombre del mensaje se reemplaza por el carácter *.

Los mensajes se numeran secuencialmente a partir de 1. Si un mensaje se envía antes de que concluya el tratamiento del precedente, es posible utilizar una numeración compuesta (ver figura 5.4) en la que el envío del mensaje 2 se produzca durante la ejecución del mensaje 1.

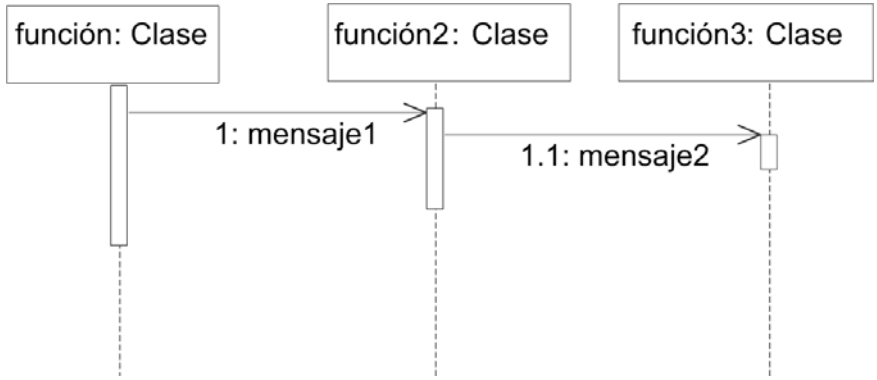


Figura 5.4 - Numeración de los mensajes

Observación

La numeración de los mensajes no es obligatoria. No obstante, resulta práctica para mostrar las activaciones anidadas.

La transmisión de datos también es posible; esta se representa mediante parámetros transmitidos con el mensaje (ver figura 5.5). El valor de cada parámetro transmitido se provee mediante el valor de variables como `dato1` y `dato2` o mediante el valor de constantes. Por defecto, el valor de los parámetros se provee en función de su orden. También es posible nombrar los parámetros para asignarles valor. El uso del carácter – significa que el valor del parámetro no se especifica. De este modo, en el ejemplo `message(-)`, no se especifica el valor de ningún parámetro.

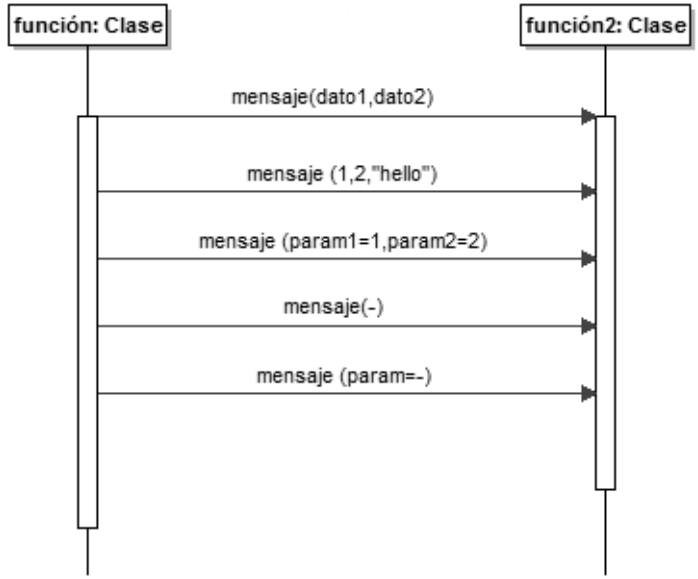


Figura 5.5 - Transmisión de datos durante el envío de un mensaje

Existen diferentes tipos de envíos de mensajes. En la figura 5.6 ofrecemos una explicación gráfica.

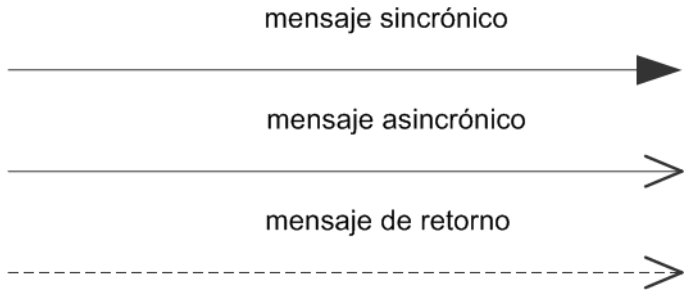


Figura 5.6 - Diferentes tipos de mensajes

El mensaje sincrónico es el utilizado con mayor frecuencia. En este caso, el expedidor espera que la activación del método mencionado por el destinatario finalice antes de continuar su actividad.

En los mensajes asincrónicos, el expedidor no espera el término de la activación invocada por el destinatario. Esto se produce al modelar sistemas en los que los objetos pueden funcionar en paralelo (es el caso de los sistemas multithreads, donde los tratamientos se efectúan en paralelo).

Ejemplo

Un jinete da una orden a su caballo, luego le da una segunda orden sin esperar a que concluya la ejecución de la primera. La primera orden constituye un ejemplo de envío de mensaje asincrónico.

El mensaje de retorno a la llamada a un método no es sistemático, ya que no todos los métodos devuelven un resultado.

La figura 5.7 ilustra los envíos de mensaje con un mensaje de retorno que transmite un resultado, bien como resultado de la función o bien como parámetro de retorno.

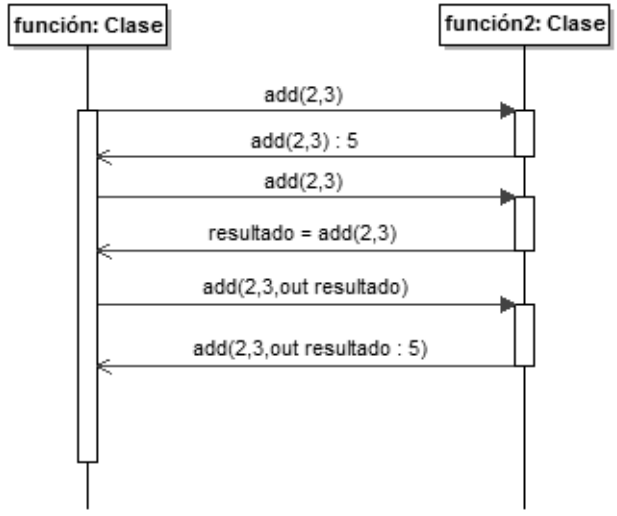
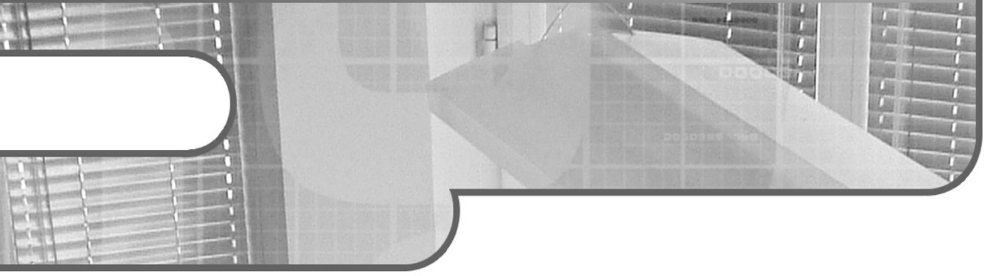


Figura 5.7 - Transmisión de resultado y mensajes de retorno



Capítulo 12
El patrón de diseño Composite

1. Descripción

El objetivo del patrón de diseño `Composite` es materializar composiciones de objetos en forma de estructura de árbol. Estas composiciones están concebidas de tal forma que un cliente tratará indiferentemente un componente y un compuesto.

2. Ejemplo

En nuestro sistema de venta de vehículos, queremos representar las empresas cliente, en especial para conocer el número de vehículos de los que disponen y proporcionarles ofertas de mantenimiento para su parque de vehículos.

Las empresas que posean filiales solicitan ofertas de mantenimiento que tengan en cuenta el parque de vehículos de sus filiales.

Una solución inmediata consiste en procesar de forma diferente las empresas sin filiales y las que posean filiales. No obstante esta diferencia en el procesado entre ambos tipos de empresa haría que la aplicación fuera más compleja y totalmente dependiente de la composición interna de las empresas cliente.

El patrón de diseño `Composite` resuelve este problema unificando ambos tipos de empresa y utilizando la composición recursiva. Esta composición recursiva es necesaria puesto que una empresa puede tener filiales que posean, ellas mismas, otras filiales. Se trata de una composición en árbol tal y como se ilustra en la figura 12.1 donde las empresas madre se sitúan sobre sus filiales. Por razones evidentes de simplificación, suponemos que no hay filiales comunes a dos empresas.

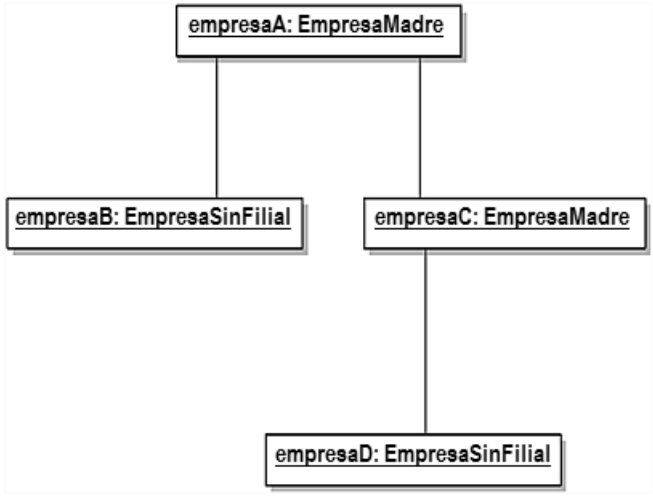


Figura 12.1 - Árbol de empresas madres y de sus filiales

La figura 12.2 muestra el diagrama de clases correspondiente. La clase abstracta `Empresa` contiene la interfaz destinada a los clientes. Posee dos subclases concretas, a saber `EmpresaSinFilial` y `EmpresaMadre`, esta última guarda una relación de agregación con la clase `Empresa` representando los enlaces con sus filiales.

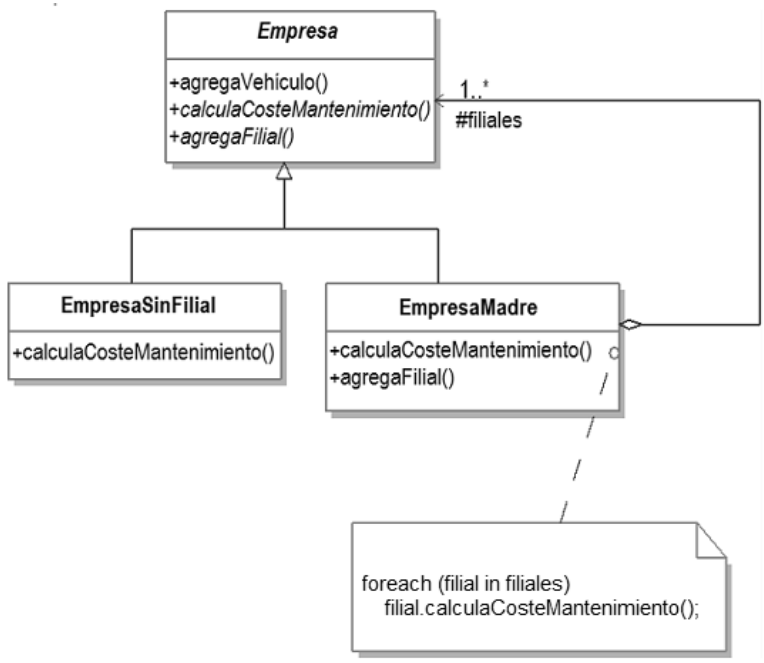


Figura 12.2 - El patrón de diseño `Composite` aplicado a la representación de empresas y sus filiales

La clase `Empresa` posee tres métodos públicos de los cuales dos son abstractos. El método concreto es el método `agregaVehiculo` que no depende de la composición en filiales de la empresa. En cuanto a los otros dos métodos, se implementan en las subclases concretas (`agregaFilial` no tiene implementación en `EmpresaSinFilial`, por eso no se representa en el diagrama de clases).

3. Estructura

3.1 Diagrama de clases

La figura 12.3 detalla la estructura genérica del patrón de diseño `Composite`.

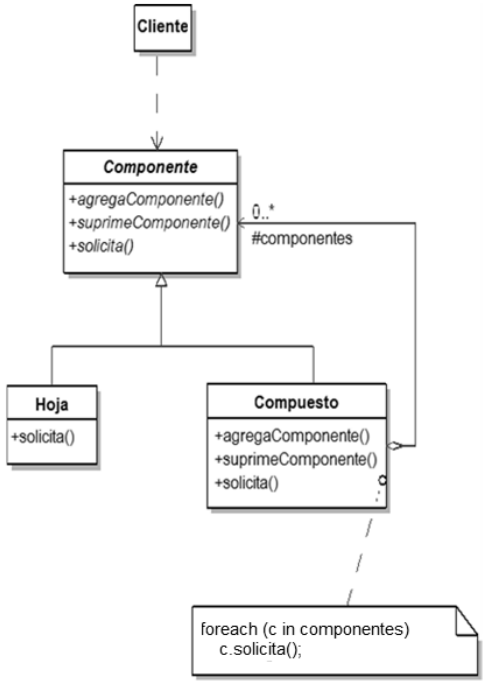


Figura 12.3 - Estructura del patrón de diseño `Composite`

3.2 Participantes

Los participantes del patrón de diseño `Composite` son los siguientes:

– `Componente` (`AbstractEmpresa`) es la clase abstracta que describe la interfaz de los objetos de la composición, implementa los métodos comunes e introduce la firma de los métodos que gestionan la composición agregando o suprimiendo componentes.

– `Hoja` (`EmpresasSinFilial`) es la clase concreta que representa las hojas de la composición (en una estructura de árbol, una hoja no posee componentes, es un nudo terminal).
– `Compuesto` (`EmpresaMadre`) es la clase concreta que representa los objetos compuestos de la jerarquía. Esta clase posee una asociación de agregación con la clase `Componente`.
– `Cliente` es la clase de los objetos que acceden a los objetos de la composición y los manipulan.

3.3 Colaboraciones

Los clientes envían sus peticiones a los componentes a través de la interfaz de la clase `Componente`.

Cuando un componente recibe una petición, reacciona en función de su clase. Si el componente es una hoja, procesa la petición tal y como se ilustra en la figura 12.4.

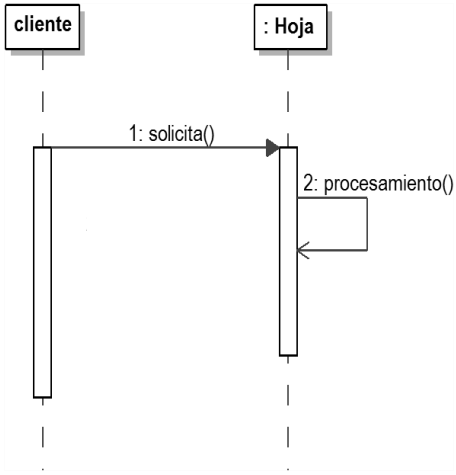


Figura 12.4 - Procesado de un mensaje por parte de una hoja

Si el componente es una instancia de la clase `Compuesto`, realiza un procesado previo, luego generalmente envía un mensaje a cada uno de sus componentes y realiza un procesamiento posterior. La figura 12.5 ilustra este comportamiento de llamada recursiva a otros componentes que van a procesar, a su vez, esta petición bien como hoja o bien como compuesto.

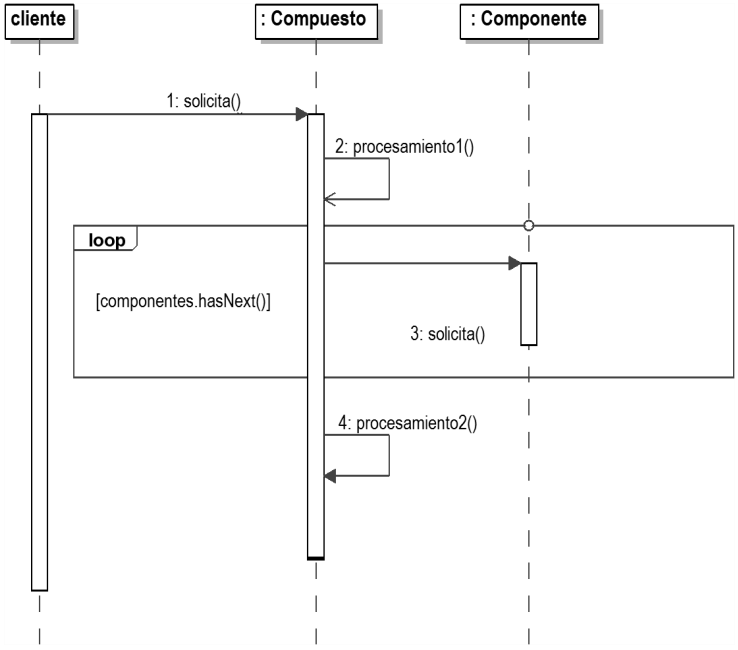


Figura 12.5 - Procesado de un mensaje por parte de un compuesto

4. Dominios de aplicación

El patrón de diseño Composite se utiliza en los siguientes casos:

– Es necesario representar jerarquías de composición en un sistema.

– Los clientes de una composición deben ignorar si se comunican con objetos compuestos o no.

5. Ejemplo en PHP

Retomemos el ejemplo de las empresas y la gestión de su parque de vehículos.

El código fuente en PHP de la clase abstracta `AbstractEmpresa` se describe a continuación. Conviene observar que el método `agregaFilial` devuelve un resultado booleano que indica si ha sido posible realizar o no la agregación.

```
<?php

declare(strict_types=1);

namespace ENI\DesignPatterns\Composite;

abstract class AbstractEmpresa
{
   protected static float $costeUnitarioVehiculo = 5;

   protected int $numVehiculos = 0;

   public function agregaVehiculo(): void
   {
       $this->numVehiculos++;
   }

   abstract public function calculaCosteMantenimiento(): float;

   abstract public function agregaFilial(AbstractEmpresa
$filial): bool;
}
?>
```

El código fuente de la clase `EmpresaSinFilial` aparece a continuación. Lógicamente, las instancias de esta clase no pueden agregar filiales.

```
<?php

declare(strict_types=1);

namespace ENI\DesignPatterns\Composite;

class EmpresaSinFilial extends AbstractEmpresa
{
   public function agregaFilial(AbstractEmpresa $filial): bool
```